APPENDIX

Installation

This appendix covers installing and setting up a Python environment for scientific computing on commonly used platforms. As discussed in Chapter 1, the scientific computing environment for Python is not a single product but rather a diverse ecosystem of packages and libraries, and there are numerous possible ways to install and configure a Python environment on any given platform. Python is rather easy to install,[1] and on many operating systems, it is even preinstalled. All pure Python libraries hosted on the *Python Package Index*[2] are also easily installed, for example, using `pip` and a command such as `pip install PACKAGE`, where `PACKAGE` is the name of the package to install. The `pip` software then searches for the package on the Python Package Index and downloads and installs it if it is found. For example, to install IPython, we can use the following command.

```
$ pip install ipython
```

And to upgrade an already installed package, add the `--upgrade` flag to the `pip` command, as follows.

```
$ pip install --upgrade ipython
```

However, many libraries for computing with Python are not pure Python libraries, and they frequently have dependencies on system libraries written in other languages, such as C and Fortran. Pip and the Python Package Index cannot handle these dependencies, and building such libraries from source requires installing C and Fortran compilers. In other words, manually installing a full scientific computing software stack for Python can take time and effort. To solve this problem, several prepackaged Python environments with automated installers have emerged. The most prominent distribution is *Anaconda*[3] which is sponsored by Anaconda Inc., a corporation with close connections and a history of supporting the open source scientific Python community. The Anaconda environment bundles the Python interpreter, the required system libraries and tools, and many scientific-computing-oriented Python libraries in an easy-to-install distribution. The following uses this Anaconda environment to set up the software to run the code discussed in this book, particularly Miniconda, a lightweight version of Anaconda, and the conda package manager.

[1] Installers for all major platforms are available for download at `www.python.org/downloads`.
[2] `http://pypi.python.org`
[3] `http://anaconda.com`

R. Johansson, *Numerical Python*, https://doi.org/10.1007/979-8-8688-0413-7

Miniconda and Conda

The Anaconda environment, which comes bundled with many libraries by default, is a convenient way to get a scientific computing environment for Python up and running quickly. However, to clarify each chapter's dependencies, we start with a Miniconda environment and explicitly install the necessary packages. In this way, we control precisely which packages are included in the environment we set up. Miniconda is a minimal version of Anaconda, which only consists of the most basic components: a Python interpreter, a few fundamental libraries, and the conda package manager. The download page for the Miniconda project (`https://conda.pydata.org/miniconda.html`) contains installers for Linux, macOS, and Windows. Download and run the installer, and follow the on-screen instructions. When the installation has finished, you should have a directory named `miniconda3` in your home directory. If you add it to your `PATH` variable during the installation, you should be able to invoke the conda package manager by running `conda` at the command prompt.

Conda[4] is a cross-platform package manager that can handle dependencies on Python packages and system tools and libraries. It is essential for installing scientific computing software, which by nature uses a diverse set of tools and libraries. Conda packages are prebuilt binaries for the target platform and are fast and convenient to install. To verify that conda is available on your system, you can use the following command.

```
$ conda --version
conda 23.10.0
```

In this case, the output tells us that conda is installed and the version is 23.10.0. To update to the latest version of conda, we can use the conda package manager itself.

```
$ conda update -n base conda
```

To update all packages installed in a particular conda environment, we can use the following command.

```
$ conda update --all
```

Once conda is installed, we can use it to install Python interpreters and libraries. We can optionally specify precise versions of the packages we want to install. The Python software ecosystem consists of many independent projects, each with their own release cycles and development goals, and there are constantly new versions of different libraries being released. This is exciting—because there is steady progress, and new features are frequently made available—but unfortunately, not all new releases of all libraries are backward compatible. It presents a dilemma for a user that requires a stable and reproducible environment over the long term and for users who simultaneously work on projects with different versions of dependencies.

The best solution in the Python ecosystem for this problem is to use a package manager such as conda to set up virtual Python environments for different projects in which different versions of the required dependencies are installed. With this approach, it is easy to maintain multiple environments with different configurations, such as separate environments for different versions of Python or environments with stable versions and development versions of required packages. I highly recommend using virtual Python environments rather than the default system-wide ones for the reasons discussed here.

With conda, new environments are created with the `conda create` command, to which we need to provide a name for the new environment using `-n NAME` or a path to where the environment is to be stored using `-p PATH`. When providing a name, the environment is, by default, stored in the `miniconda3/envs/NAME`

[4] `http://conda.pydata.org/docs/index.html`

directory. When creating a new environment, we can also list packages to install. At least one package must be specified. For example, to create two new environments based on Python 3.10 and Python 3.11, we can use the following commands.

```
$ conda create -n py3.10 python=3.10
$ conda create -n py3.11 python=3.11
```

The environments are named `py3.10` and `py3.11`. To use one of these environments, we need to *activate* it using the `conda activate py3.10` command or the `conda activate py3.11` command, respectively. To *deactivate* an environment, use `conda deactivate`. With this method, it is easy to switch between different environments, as illustrated in the following sequence of commands.

```
$ conda activate py3.10
(py3.10)$ python --version
Python 3.10.13
(py3.10)$ conda activate py3.11
(py3.11)$ conda --version
Python 3.11.5
(py3.11)$ conda deactivate
$
```

The `conda env`, `conda info`, and `conda list` commands are helpful tools to manage environments. The `conda info` command can list available environments (same as `conda env list`).

```
$ conda info --envs
# conda environments:
#
base                  /Users/rob/miniconda3
py3.10                /Users/rob/miniconda3/envs/py3.10
py3.11                /Users/rob/miniconda3/envs/py3.11
```

The `conda list` command can list installed packages and their versions in a given environment.

```
$ conda list -n py3.10
# packages in environment at /Users/rob/miniconda3/envs/py3.10:
#
# Name                    Version                   Build  Channel
bzip2                     1.0.8                h1de35cc_0
ca-certificates           2023.08.22           hecd8cb5_0
libffi                    3.4.4                hecd8cb5_0
ncurses                   6.4                  hcec6c5f_0
openssl                   3.0.12               hca72f7f_0
pip                       23.3.1          py310hecd8cb5_0
python                    3.10.13              h5ee71fb_0
readline                  8.2                  hca72f7f_0
setuptools                68.0.0          py310hecd8cb5_0
sqlite                    3.41.2               h6c40b1e_0
tk                        8.6.12               h5d9f67b_0
tzdata                    2023c                h04d1e81_0
```

```
wheel                     0.41.2          py310hecd8cb5_0
xz                        5.4.2                h6c40b1e_0
zlib                      1.2.13               h4dc903c_0
```

The `conda env export` command produces similar information in YAML format[5].

```
(py3.10)$ conda env export

name: py3.10
channels:
  - defaults
dependencies:
  - bzip2=1.0.8=h1de35cc_0
  - ca-certificates=2023.08.22=hecd8cb5_0
  - libffi=3.4.4=hecd8cb5_0
  - ncurses=6.4=hcec6c5f_0
  - openssl=3.0.12=hca72f7f_0
  - pip=23.3.1=py310hecd8cb5_0
  - python=3.10.13=h5ee71fb_0
  - readline=8.2=hca72f7f_0
  - setuptools=68.0.0=py310hecd8cb5_0
  - sqlite=3.41.2=h6c40b1e_0
  - tk=8.6.12=h5d9f67b_0
  - tzdata=2023c=h04d1e81_0
  - wheel=0.41.2=py310hecd8cb5_0
  - xz=5.4.2=h6c40b1e_0
  - zlib=1.2.13=h4dc903c_0
prefix: /Users/rob/miniconda3/envs/py3.10
```

To install additional packages in an environment, we can specify a list of packages when the environment is created or activate the environment and use `conda install` or the `conda install` command with the `-n` flag to specify a target environment for the installation. For example, to create a Python 3.11 environment with NumPy version 1.24, we could use the following command.

```
$ conda create -n py3.11-np1.24 python=3.11 numpy=1.24
```

To verify that the new environment `py3.11-np1.24` contains NumPy of the specified version, we can use the `conda list` command again.

```
$ conda list -n py3.11-np1.24
# packages in environment at /Users/rob/miniconda3/envs/py3.11-np1.24:
#
# Name                    Version                   Build  Channel
blas                      1.0                         mkl
bzip2                     1.0.8                h1de35cc_0
ca-certificates           2023.08.22           hecd8cb5_0
intel-openmp              2023.1.0         ha357a0b_43548
libcxx                    14.0.6               h9765a3e_0
libffi                    3.4.4                hecd8cb5_0
```

[5] http://yaml.org

```
mkl                       2023.1.0         h8e150cf_43560
mkl-service               2.4.0           py311h6c40b1e_1
mkl_fft                   1.3.8           py311h6c40b1e_0
mkl_random                1.2.4           py311ha357a0b_0
ncurses                   6.4                  hcec6c5f_0
numpy                     1.24.3          py311h728a8a3_1
numpy-base                1.24.3          py311h53bf9ac_1
openssl                   3.0.12               hca72f7f_0
pip                       23.3.1          py311hecd8cb5_0
python                    3.11.5               hf27a42d_0
readline                  8.2                  hca72f7f_0
setuptools                68.0.0          py311hecd8cb5_0
sqlite                    3.41.2               h6c40b1e_0
tbb                       2021.8.0             ha357a0b_0
tk                        8.6.12               h5d9f67b_0
tzdata                    2023c                h04d1e81_0
wheel                     0.41.2          py311hecd8cb5_0
xz                        5.4.2                h6c40b1e_0
zlib                      1.2.13               h4dc903c_0
```

Here, we see that NumPy is indeed installed, and the precise version of the library is 1.24.3. If we do not explicitly specify the version of a library, the latest stable release is used.

To use the second method—to install additional packages in an already existing environment—we first activate the environment.

```
$ conda activate py3.11
```

Then use `conda install PACKAGE` to install the package with the name `PACKAGE`. Here, we can also give a list of package names. For example, to install the NumPy, SciPy, and Matplotlib libraries, we can use the following command.

```
(py3.11)$ conda install numpy scipy matplotlib
```

Or, we could also use the following command.

```
$ conda install -n py3.11 numpy scipy matplotlib
```

When installing packages using conda, all required dependencies are also installed automatically, and the preceding command also installed the packages `dateutil`, `freetype`, and `libpng`, among others, which are dependencies for the `matplotlib` package.

```
(py3.11)$ conda list | grep -e matplotlib -e dateutil -e freetype -e libpng
freetype                  2.12.1               hd8bbffd_0
libpng                    1.6.39               h6c40b1e_0
matplotlib                3.8.0           py311hecd8cb5_0
matplotlib-base           3.8.0           py311h41a4f6b_0
python-dateutil           2.8.2              pyhd3eb1b0_0
```

Note that not all the packages installed in this environment are Python libraries. For example, `libpng` and `freetype` are system libraries, but conda can handle and install them automatically as dependencies. This is one of the strengths of conda compared to, for example, the Python-centric package manager `pip`.

To update selected packages in an environment, we can use the `conda update` command. For example, to update NumPy and SciPy in the currently active environment, we can use the following command.

```
(py3.11)$ conda update numpy scipy
```

To remove a package, we can use `conda remove PACKAGE`. To completely remove an environment, we can use `conda remove -n NAME --all`. For example, to remove the environment `py3.11-np1.24`, we can use the following command.

```
$ conda remove -n py3.11-np1.24 --all
```

Conda locally caches packages that have once been installed. This makes it fast to reinstall a package in a new environment and quick and easy to tear down and set up new environments for testing and trying out different things without any risk of breaking environments used for other projects. To re-create a conda environment, all we need to do is to keep track of the installed packages. Using the `-e` flag with the `conda list` command gives a list of packages and their versions in a format compatible with the `pip` software. This list can be used to replicate a conda environment, for example, on another system or at a later point in time.

```
$ conda list -e > requirements.txt
```

With the `requirements.txt` file, we can now update an existing conda environment in the following manner.

```
$ conda install --file requirements.txt
```

Or create a new environment that is a replication of the environment that was used to create the `requirement.txt` file.

```
$ conda create –n NAME --file requirements.txt
```

Alternatively, we can use the YAML format dump of an environment produced by `conda env export`.

```
$ conda env export –n NAME > env.yml
```

In this case, we can reproduce the environment using the following.

```
$ conda env create --file env.yml
```

Note that here we do not need to specify the environment name since the `env.yml` file also contains this information. This method also has the advantage that packages installed using `pip` are installed when the environment is replicated or restored.

Jupyter Notebook Kernel Registration

It is a good practice to frequently create new environments for new projects, testing upgrades of external dependencies, and so on. When we work with the Jupyter Notebook environment, as introduced in Chapter 1, we need to register a new kernel for each new environment we create to be available in the kernel list in the Jupyter web application. To do this, we can activate the environment, install the `ipykernel` package, and run the command for registering the currently active.

```
$ conda activate py3.11
(py3.11)$ conda install ipykernel
(py3.11)$ python -m ipykernel install --user --name py3.11
Installed kernelspec py3.11 in /Users/rob/Library/Jupyter/kernels/py3.11
```

A Complete Environment

Now that we have explored the conda package manager and seen how it can be used to set up environments and install packages, let's go over the procedures for setting up a complete environment with all the dependencies required for the material in this book. The following uses the `py3.10` environment, which was previously created using the following command.

```
$ conda create –n py3.10 python=3.10
```

This environment can be activated using the following.

```
$ conda activate py3.10
```

Once the target environment is activated, we can install the libraries used in this book with the following commands.

```
conda install ipython jupyter jupyterlab spyder pylint pyflakes pep8
conda install numpy scipy sympy matplotlib networkx pandas seaborn
conda install patsy statsmodels scikit-learn pymc
conda install h5py pytables msgpack-python cython numba cvxopt pyarrow
conda install -c conda-forge fenics mshr
conda install -c conda-forge pygraphviz
pip install scikit-monaco
pip install version_information
```

The FEniCS libraries have many intricate dependencies, making installing this standard approach on some platforms difficult.[6] For this reason, if the FEniCS installation using conda fails, it is most easily installed using the prebuilt environments available from the project's website: https://fenicsproject.org/download. Another good solution for obtaining a complete FEniCS environment can be to use a Docker[7] container with FEniCS preinstalled. See, for example, https://registry.hub.docker.com/repos/fenicsproject for more information about this method.

Table A-1 presents a breakdown of the installation commands for the dependencies on a chapter-by-chapter basis.

[6] There are recent efforts to create conda packages for the FEniCS libraries and their dependencies: http://fenicsproject.org/download/.

[7] For more information about software container solution Docker, see www.docker.com.

Table A-1. *Installation Instructions for Dependencies for Each Chapter*

Chapter	Used Libraries	Installation

1	IPython, Spyder, Jupyter	`conda install ipython jupyter jupyterlab`
		`conda install spyder pylint pyflakes pep8`
		Here, `pylint`, `pyflakes`, and `pep8` are code analysis tools that Spyder can use.
		To convert IPython notebooks to PDF, you also need a working LaTeX installation.
		To book-keep which versions of libraries were used to execute the IPython notebooks that accompany this book, the IPython extension command `%version_information` was used; it is available in the `version_information` package that can be installed with `pip`:
		`pip install version_information`.

2	NumPy	`conda install numpy`
3	NumPy, SymPy	`conda install numpy sympy`
4	NumPy, Matplotlib	`conda install numpy matplotlib`
5	NumPy, SymPy, SciPy, Matplotlib	`conda install numpy sympy scipy matplotlib`
6	NumPy, SymPy, SciPy, Matplotlib, cvxopt	`conda install numpy sympy scipy matplotlib cvxopt`
7	NumPy, SciPy, Matplotlib	`conda install numpy scipy matplotlib`

8	NumPy, SymPy, SciPy, Matplotlib, Scikit-Monaco	`conda install numpy sympy scipy matplotlib`
		There is no conda package for scikit-monaco, so we need to install this library using `pip`:
		`pip install scikit-monaco`
		At the time of writing, this library is also not compatible with the latest version of Python, and to use this library, it might be necessary to use an older version of Python, such as Python 3.6.

9	NumPy, SymPy, SciPy, Matplotlib	`conda install numpy sympy scipy matplotlib`

10	NumPy, SciPy, Matplotlib, NetworkX	`conda install numpy scipy matplotlib networkx`
		To visualize NetworkX graphs, we also need the Graphviz library (see `www.graphviz.org`) and its Python bindings in the `pygraphviz` library:
		`conda install pygraphviz`

11	NumPy, SciPy, Matplotlib, and FEniCS	`conda install numpy scipy matplotlib` `conda install -c conda-forge fenics mshr`
12	NumPy, Pandas, Matplotlib, Seaborn	`conda install numpy pandas matplotlib seaborn`

(*continued*)

Table A-1. (*continued*)

Chapter	Used Libraries	Installation
13	NumPy, SciPy, Matplotlib, Seaborn	`conda install numpy scipy matplotlib seaborn`
14	NumPy, Pandas, Matplotlib, Seaborn, Patsy, Statsmodels	`conda install numpy pandas matplotlib seaborn patsy statsmodels`
15	NumPy, Matplotlib, Seaborn, scikit-learn	`conda install numpy matplotlib seaborn scikit-learn`
16	NumPy, Matplotlib, PyMC	`conda install numpy matplotlib pymc`
17	NumPy, SciPy, Matplotlib	`conda install numpy scipy matplotlib`

18	NumPy, Pandas, h5py, PyTables, msgpack, PyArrow	`conda install numpy pandas h5py pytables msgpack-python pyarrow`
		At the time of writing, the `msgpack-python` conda package is not available for all platforms. When conda packages are not available, the msgpack library needs to be installed manually, and its Python bindings can be installed using `pip`:
		`pip install msgpack-python`

19	NumPy, Matplotlib, Cython, Numba	`conda install numpy matplotlib cython numba`

A list of the packages and their exact versions used to run the code included in this book is also available in the `requirements.txt` file that is available for download together with the code listing. With this file, we can create an environment with all the required dependencies with a single command.

```
$ conda create -n py3.10 --file py3.10_requirements.txt
```

Alternatively, we can re-create the `py3.10` and `py3.11` environments using the exports `py3.10-env.yml` and `py3.11-env.yml`. These files are also available together with the source code listings.

```
$ conda env create --file py3.10-env.yml
$ conda env create --file py3.11-env.yml
```

Summary

This appendix reviewed the installation of the various Python libraries used in this book. The Python environment for scientific computing is not a monolithic environment; rather, it consists of an ecosystem of diverse libraries maintained and developed by different groups of people, following different release cycles and development paces. Consequently, collecting all the necessary pieces of a productive setup from scratch can be difficult. In response to this problem, several solutions addressing this situation have appeared, typically in the form of prepackaged Python distributions. Anaconda is a popular example of such a solution in the Python scientific computing community. Here, I focused on the conda package manager from the Anaconda Python distribution, which, in addition to being a package manager, also allows the creation and management of virtual installation environments.

Further Reading

If we want to create Python source packages for your own projects, see, for example, `https://packaging.python.org/en/latest/index.html`. Particularly study the `setuptools` library and its documentation at `https://github.com/pypa/setuptools`. Using `setuptools`, we can create installable and distributable Python source packages. Once a source package has been created using `setuptools`, creating binary conda packages for distribution is usually straightforward. For information on creating and distributing conda packages, see `https://docs.conda.io/projects/conda-build/en/stable/user-guide/tutorials/build-pkgs.html`. See also the `conda-recipes` repository at `github.com`, which contains many examples of conda packages: `http://github.com/conda/conda-recipes`. Finally, `www.anaconda.org` is a conda package hosting service with many public channels (repositories) where custom-built conda packages can be published and installed directly using the conda package manager. Many packages unavailable in the standard Anaconda channel can be found on user-contributed channels on `anaconda.org`. Many packages are available in the `conda-forge` channel, built from conda recipes on `conda-forge.org`.